SPARC の特徴を生かした UtiLisp/C の実現法†

田中 哲朗††

この論文では，SPARC プロセッサの上で実行するために開発した UtiLisp/C の実現法について，型チェックとエラー処理を中心に述べる．Lisp の実行には動的な型チェックが必要である．型チェックは Lisp の実行時間のかなりの部分を占めるので，高速な Lisp 処理系を作るには効率的な型チェックを実現しなければならない．実行速度を重視して言語が設計されている UtiLisp は，特にこの点を考慮して処理系が作られてきた．今回，SPARC プロセッサ上の処理系を作るにあたっても，そのことに留意して SPARC プロセッサ固有の機能を使った高速な型チェック機構を実現した．そのため，高級言語で書いたにもかかわらず十分な実行速度を得ることができた．

1. はじめに

UtiLisp は Lisp 言語の一方言であり，実行速度を重視して設計されている[1),2)]．

最初の処理系はメインフレームのアセンブリ言語で記述された．他のプロセッサへの移植は難しいと思われたが，最初に汎用レジスタの数やアドレス空間の広さなどメインフレームと多くの共通点を持つ MC 68000 への移植 (UtiLisp 68) が実現した[3)]．

その後，これらのプロセッサと違って 32 ビットすべてをアドレス指定に用いるプロセッサが主流となった．このようなプロセッサではそれまでの UtiLisp で用いていた型チェックやヒープ管理法を用いることができない．そこで，言語仕様を一部拡張しデータ表現を作り直した UtiLisp 32 という処理系が作られた．

UtiLisp 32 は生のアセンブリ言語ではなく LAP 形式というS式の形で書かれ，Lisp でマクロ処理を行ってアセンブリ言語のソースを出力することによって，プロセッサの違いを容易に吸収することができ，MC 68010/20，VAX などのいくつかのプロセッサ上で実現された[4)]．

近年になって RISC アーキテクチャに基づくプロセッサが登場し，32 ビットプロセッサの主流になった．これらのプロセッサは高級言語（主にC言語）で書かれたプログラムを高速に実行するように命令セットが設計されている．そのため，高級言語でプログラムを書いても，人間がアセンブリ言語で巧妙に書いたプログラムとの実行時間の差はそれほど大きくないと思われる．

そこで，RISC プロセッサの一つである SPARC プロセッサ用の UtiLisp 処理系を作るにあたって，記述言語はこれまでの処理系のようにアセンブリ言語ではなく，デバッグの容易さや移植性を重視してC言語とすることに決めた．また，UtiLisp 32 の型チェック法はC言語では効率的に実現することができないため，C言語で容易に書け，SPARC プロセッサの特徴を生かした別の型チェック法を用いることにした．

このような方針によって作られた処理系が UtiLisp/C である．UtiLisp/C に関しては SPARC のアセンブリ言語の特徴を生かした bignum ルーチンや[6)]，各種のオプティマイズを行っているコンパイラ[8)]などさまざまな話題があるが，この論文では型の表現，メモリ管理，エラー処理に話題をしぼって述べる．

2. 型の表現

2.1 旧 UtiLisp における型表現

メインフレーム版の UtiLisp および MC 68000 用の UtiLisp 68 では，アドレスとして 24 ビット分しか使われないことを利用して，32 ビット中の上位 8 ビットをオブジェクトの型を表すタグとして使った．その際タグの割当を巧妙に行ったため，型チェックを高速に行うことができた．

UtiLisp 32 では，アドレス空間が 32 ビットに広がり，この方法を適用することができなくなった．アドレス空間の下位 24 ビットだけを使うことを前提にしてこの方法を使うことも可能だが，その場合メモリの参照のたびにタグ部分をマスクする命令を書かなくてはいけない．これは効率に大きく影響する．そこで，新たな型表現を考え出すことになった．

† UtiLisp/C: An Implementation of UtiLisp Featuring SPARC Architecture by TETSUROU TANAKA (Information Engineering Course, Graduate School of Engineering, University of Tokyo).

†† 東京大学大学院工学系研究科情報工学専攻

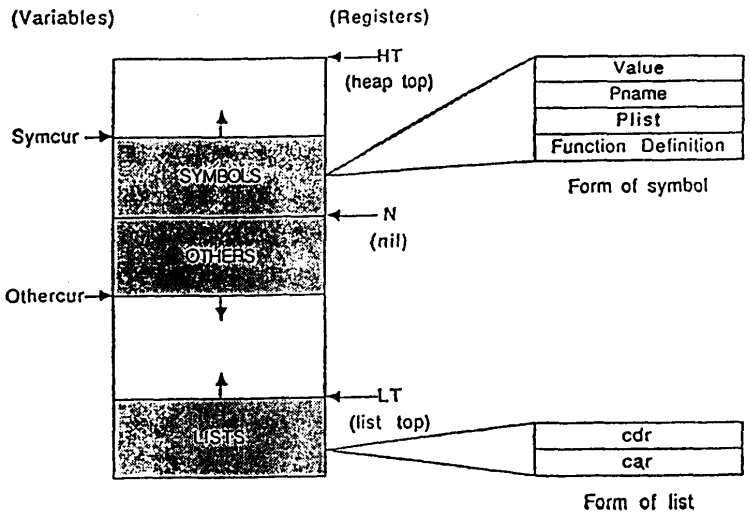

図 1 UtiLisp 32 のヒープ構造
Fig. 1 Heap allocation of UtiLisp 32.

UtiLisp 32 ではヒープ分割法を使った．これは，シンボル (symbol)，コンス (cons)，それ以外のヒープ上に実体を持つオブジェクト (others) をヒープ上の別々の領域に置き（**図 1**），アドレスの大小の比較によって型を決定する方法である．シンボルとコンスはこの方法だけで型を決定することができる．

ヒープ上に実体を持たない固定長整数は，ヒープとは絶対に重ならないよう上位 2 ビットにタグを付け，さらに GC に使う下位 2 ビットを除いた 28 ビットに値を入れる．others にはさらに型を区別するためにオブジェクトの先頭に 1 ワードのオブジェクトタグを付ける．UtiLisp/C でも UtiLisp 32 の型表現法を継承することは考えられたが，C言語では領域の境界のアドレス（広域変数）をレジスタに入れておくことができず，型チェックのたびにメモリ参照が生じてしまい，実行速度が大きく低下する．そのため，別の方法を取ることにした．

2.2 下位ビットをタグとする方法

UtiLisp ではヒープをワード（4 バイト）単位で確保するから，オブジェクトの先頭アドレスの下位 2 ビットは必ず 00 になる．この部分をタグとして扱えば 4 種類の型を表現できる．

この方法は，ポインタの上位にタグを付ける場合と違い，タグ部分を容易に取り出すことができ，メモリを参照する時はマスクを取るかわりにタグを打ち消す小さなオフセットを付けるだけでよいという点で優れている．C言語で記述する際もマクロを使えば低レベルの詳細を意識しないで書くことができる．これらの利点を重視し UtiLisp/C ではこの方法を採用することにした．

この方法は UtiLisp 32 の設計の際にも考慮されたが，採用に至らなかった．その方法を UtiLisp/C で採用したのは，RISC プロセッサ上で動かすことを想定しているためである．

CISC プロセッサではメモリ参照の際にオフセットが付いていると，オフセットがない場合と比べて命令の実行時間が長くなる．また，オフセット付きの場合は使えないアドレッシングモードもあるため（オートインクリメントなど），オフセットがなければ，一命令ですむ時に，複数の命令が必要になることがある．これではマスクを取る代わりにオフセットを付けて参照できるという利点を生かすことができない．

その点，RISC プロセッサはアドレッシングモードが少なく，オフセットが付いている場合とない場合とで実行時間に差がないものが多い．そのため，下位ビットをタグとして使っても，タグが付いていない場合との参照時間の差は生じない．

さらに，SPARC や一部の RISC プロセッサのようにワード境界をまたがったメモリ参照を許さないプロセッサでは都合がよいことがある．

通常，引数として特定の型しか許さない組込み関数では，実行時に引数が然るべき型かをチェックし，結果に応じてエラー処理ルーチンに分岐するようなコードを入れておく必要がある．

しかし，SPARC のようなプロセッサの場合は，オブジェクトの内容を参照する際の副作用として型チェックが行われるので型チェックのためのコードを入れる必要がない．想定していない型の引数の場合はタグを打ち消すオフセットを付けたアドレスが，ワード境界とずれてしまい，メモリ参照でトラップが生じ，ここで型エラーが検出されるからである．

2.3 UtiLisp/C の型表現

決定した UtiLisp/C の型表現を**表 1** に示す．

下位 2 ビットだけで区別することのできるのは 4 種類のオブジェクトである．UtiLisp の型すべてを表現

表 1 UtiLisp/C の型表現
Table 1 Internal representation of Lisp objects.

型	ポインタタグ	オブジェクトタグ (32 bit 中下位 6 bit)
固定長整数	0000	
シンボル	01	
コンス	10	
浮動小数点実数	11	000100
可変長整数	11	001100
配列	11	011100
文字列	11	100100
ストリーム	11	101100
コード	11	110100

することができないため，使用頻度の小さいオブジェクトではオブジェクトタグも併用する．

UtiLisp 32 にならって，ポインタタグだけで判別することのできる型は，固定長整数，シンボル，コンスの三つとした．

固定長整数はメモリ上に実体がないためワード境界トラップを利用した型チェック法を使うことができないが，SPARC にタグ付き加減算命令というのがあるのでそれを利用した型チェックを行う．SPARC のタグ付き加減算命令とは，オペランドのどちらかの下位2ビットが 00 でない時はトラップを生ずるという命令である．この命令のため固定長整数のタグは自動的に 00 に決まる．

固定長整数は下位2ビットに加えてさらに2ビット分をタグとして使う．これは GC 実行中にオブジェクトタグと区別するために1ビット必要であり，さらに1ビット削って UtiLisp 32 と精度を互換にしたためである．

3. メモリ管理

UtiLisp 32 はヒープ分割法を用いている．ヒープは三つに分かれており，シンボル，コンス，others がそれぞれ別の領域を占めている．それに対し，UtiLisp/C では，前章で述べたような型表現を用いているため，ヒープは一つにまとめられて，いろいろな型のオブジェクトが混在している．

UtiLisp の処理系では，高速化のためヒープ以外にスタック上にも Lisp オブジェクトを置くようにしている．これまで作られた UtiLisp の処理系はいずれもハードウェアのスタックをそのまま利用して1本のスタックに，サブルーチンの戻り番地や Lisp の引数，束縛情報などを混在して置いていた．

UtiLisp/C はCで記述したため，スタックの中身を全部自由に管理することができなくなった．Cで書かれた多くの Lisp 処理系は，Lisp オブジェクトをハードウェアスタックに入れておくために，スタック上にリンク構造を作るようにしている．これでは，スタックを使う目的の一つである高速性が失われてしまうので，Lisp オブジェクトだけを収めるスタックをソフトウェアで別途用意することにした．

UtiLisp/C ではスタックに積むオブジェクトの性格を4通りに分け，引数，バインド，関数，環境と四つのスタックを用意することにした（図 2）．

引数スタックは関数への引数を積むためのスタックで，他に eval や組込み関数の中で，GC の際に保存しておくローカル変数を保持するためにも使われる．このスタックは頻繁に使われるのでスタックポインタはアクセスの遅いグローバル変数としてではなく，Cの関数の引数として渡す．

バインドスタックには，lambda バインドをする際にシンボルとシンボルの値を組にして積んで置く．このスタックのスタックポインタは，アクセスの遅いグローバル変数だが，続けて参照する時には，一時的にローカルなレジスタ変数にコピーして使うことによって，実行の高速化を図っている．

関数スタックは，エラーが生じた時にどこでエラーが生じたかを表示したり，backtrace という関数で実行時の関数の呼び出し関係を参照するのに使う．コンパイルされたコードの場合は，関数スタックの先頭の内容が実行中の関数のコードを指していることを利用し，このスタックを介してリテラルの参照を行う．

環境スタックは，break と unbreak, catch と throw, loop と exit, prog と go, return などの大域脱出用プリミティブ対をサポートするために使われ

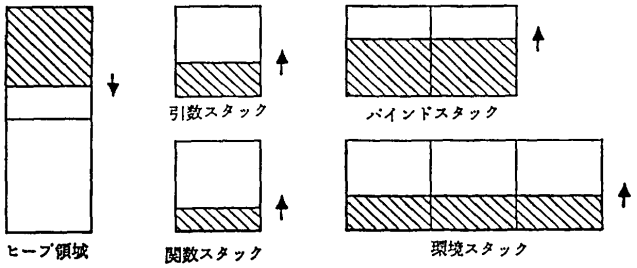

図 2 UtiLisp/C のメモリマップ
Fig. 2 Memory map of UtiLisp/C.

る.

UtiLisp/C はC言語で書かれているので UtiLisp 32 のようにコントロールスタックを操作することができず，大域脱出の際に，スタックの内容をしらべてコンテクストの回復を行うことができない．あらかじめ setjmp を行っておいて脱出時に longjmp を行う．環境スタックの三つ組の一つには，jmp_buf へのポインタを入れておく．

このほかに，nil や t など組込み関数から参照されるシンボルはルートオブジェクトとして別のところに置く．

4. ゴミ集め (GC)

UtiLisp/C ではコピー方式の GC を採用している．コピー方式の GC を用いるとヒープ領域の使用量が倍になるが，GC のアルゴリズムは単純になり，結果として GC にかかる時間も少なくなる．

コピー方式の GC を自然に記述すると再帰的に書ける．再帰的にコピーしていくと参照が局所化するという利点があるが，最悪の場合はヒープ領域の大きさに比例した回数の再帰呼び出しが生じて大量のスタックを消費する．これほど極端な場合でなくても SPARC の場合再帰呼び出しのレベルが深い時のオーバヘッドが大きいので，GC の速度低下を引き起こしかねない．

UtiLisp/C では再帰を使わないコピー方式の GC のアルゴリズムを採用した．これは，古いヒープから新しいヒープに Lisp オブジェクトをコピーする際に，その Lisp オブジェクトからの参照をその場で更新しないで，後で新しい領域を走査した時に初めて更新するという方法である．

この方法の場合，新しいヒープを走査する際に先頭ワードを見てそのオブジェクトの型が決定できなければいけない．オブジェクトタグが付いていないコンスやシンボルは，ここに古いオブジェクトへのポインタを書いておいてこの時点でコピーするようにした．

ヒープの残りが少なくなってくると，頻繁に GC が起き，実行速度が低下する．そのため，UtiLisp/C では GC 後のフリー領域がヒープ全体に占める割合が，ある程度小さくなったら，ヒープを倍の大きさに確保し直して実行を再開するという機能を付けた．

5. 組込み関数

組込み関数の中には，可変個の引数をもつものがある．そのためこれまでの UtiLisp の処理系では，組込み関数のエントリーポイントを複数用意し，引数の数に応じて別のアドレスから関数の実行を始めるようにしていた．

しかし，C言語の関数のエントリーは一つしかなく，この方法を使うと引数の数ごとに似たような別の関数を作らなければいけなくなる．これは実行ファイルのサイズを大きくしデバッグを困難にする．そこで UtiLisp/C ではCの関数の引数として Lisp の引数の数を与え，関数の中で自分で引数の数をチェックして分岐するようにした．

組込み関数の記述の例として関数 car の定義を示す．

```
WORD car_f (na, fp)
WORD *fp;
{
  WORD a;

  if (na!=1) parerr ( );
  return (car (checkcons (a, ag (0))));
}
```

引数 na は Lisp の引数の数で，fp は引数スタックのスタックポインタである．parerr, car, checkcons, ag はいずれもCのマクロであり，プロセッサの違いをここで吸収する．

SPARC では固定長整数を扱う関数の中でタグ付き加減算命令を使うが，これはC言語で書くことができず，インライン関数を使うことにした．Sun 標準のCコンパイラの場合は，

```
.inline _tadd, 8
taddcctv %o0, %o1, %o0
.end
```

のように定義したファイルを作っておき，コンパイル時に指定すると，tadd という名前の関数があるかのように扱うことができる．

6. エラー処理

通常のエラー呼び出しは関数呼び出しによって行うが，signal によってエラー処理を行う場合もある．この場合エラーを検出した後，エラー処理を行う際のいろいろな工夫が必要になる．以下にそれらの場合処理について述べる．

6.1 型エラー (1)

固定長整数以外の型エラーはすべてワード境界をま

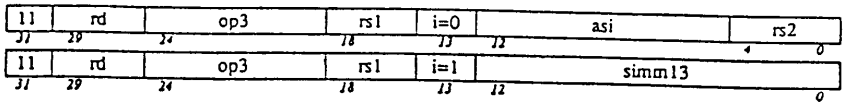

図 3 ロード/ストア命令のフォーマット
Fig. 3 Format of load/store instructions.

たがったワードアクセスによるバスエラーによって検出する．バスエラーが生ずると，SIGBUS の signal が発生する．

signal ハンドラには，OS に依存する形で signal 発生時の種々の情報が渡されるが，そこから UtiLisp のエラー処理関数にエラーの生じた場所と，エラーの原因のオブジェクトを引数として渡す必要がある．

例えば，a というシンボルに対して car を実行しようとして生じた型エラーでは a と C#car (car のコード) を引数として err:argument-type を funcall しなければならない．

エラーの生じた場所は，関数スタックを見ればわかる．後はエラーの原因となった Lisp オブジェクトを決定することができればよい．

Sun 4 では，ワード境界を原因とするバスエラーが生じた時，signal ハンドラに，エラーを生じたアドレスの情報が渡ってこない．UtiLisp/C では SPARC が RISC であることを利用して，そのアドレスを決定している．

まず，signal 発生時の PC の内容から，どの命令を実行しようとして，エラーが発生したかが分かる．バスエラーを生ずる以上，その命令はロード命令かストア命令である．SPARC は RISC なので，アドレッシングモードは，レジスタ＋オフセットとレジスタ＋レジスタしかない (**図 3**)．

Lisp オブジェクトの内容を参照する場合に使われるのはレジスタ＋オフセットのアドレッシングモードだけである．その場合，signal 発生時に参照に使われたレジスタには，Lisp オブジェクトそのものが入っている．後はそのレジスタの保存された値を引数にしてエラー処理関数を呼び出せばよい．

6.2 型エラー (2)

タグ付き加減算命令 taddcctv, tsubcctv は，両方のオペランドの下位 2 ビットがともに 00 でない時にエラーを起こすが，両方とも 00 であっても演算の結果，オーバフローが生ずると同様にエラーを生ずる．

signal ハンドラは，足そうとしたレジスタ (片方が即値のこともある) の内容を調べ，どちらかのタグが 00 でない時は，その内容をもってエラー処理関数を funcall する．両方が 00 でない時，つまり複数の引数について型エラーが生じた時，どちらのエラーを表示するかは，処理系製作の自由度として残されている．タグが両方とも 00 の時は，オーバフローと分かる．UtiLisp では固定長整数のオーバフローはエラーにしていないため，そのままリターンすればよい．

6.3 引数チェック

UtiLisp/C ではエラー関数はほとんどの組込み関数の中から呼ばれる可能性がある．大部分の組込み関数は他の C の関数を呼び出さない末端関数である．Sun 4 用の C コンパイラは末端関数については，レジスタウィンドウをずらさないで，手持ちのレジスタだけでやりくりする高速なコードを出してくれるが，エラー関数を呼び出す可能性があるために末端関数ではなくなってしまうと，効率の低下につながる．

そこで，引数の数が違うというエラーも signal を使って処理する．引数の数が違う時は，特定の必ずバスエラーを起こすアドレスをアクセスしてトラップを生じさせてエラー処理ルーチンに飛ぶのである．末端関数にすることのメリットがない場合は通常どおり関数呼び出しによってエラー処理を行う．

7. 評 価

7.1 実行速度

実行速度の評価は同じプロセッサ上の他の処理系との比較で行うのが一般的だが，同じ Lisp 処理系とはいっても言語仕様の違いによってインタプリタの速度が大きく違ってくるので比較にならない．UtiLisp 32 よりは遅くならないというのが一つの目標だったので，UtiLisp 32 と比較することにする．

ベンチマークの結果を**表 2** に示す．ベンチマークに使用したのは，Lisp コンテストで使われた問題で，Lisp のベンチマークでは良く使われている．

16.7 MHz の MC 68020 を使った公称 1.5 MIPS の Sun 3/50 と 20 MHz の SPARC CPU を使った公称 12.5 MIPS の Sparc Station 1 (SS 1) という能力に大きな差のある計算機を使っただけに，これだけ見て UtiLisp/C の設計方針の正当性を主張することはできない．しかし，当初の方針どおり，どの問題で

表 2 ベンチマーク結果
Table 2 Benchmarks.

番号	プログラム名	UtiLisp 32 (Sun 3/50) 1/60 sec	UtiLisp/C (SS 1) 1/60 sec	時間比
1-1	tarai	1183	417	0.35
1-4	tak	658	233	0.35
2-1	list-tarai	205	93	0.45
2-2	srev	651	298	0.46
2-4	qsort	685	317	0.46
2-5	nrev	501	303	0.60
2-6	reverse	83 (184)	48 (42)	0.58
2-7	nreverse	42 (182)	22 (42)	0.52
3-1	string-tarai	2296 (462)	898 (139)	0.39
4-1	flo-tarai	1431 (143)	472 (34)	0.33
4-2	big-tarai	1664 (84)	710 (18)	0.43
5-1	bubblesort	284	152	0.54
6-1	sequence	52	19	0.37
7-1	(bita)	141	93	0.66
7-3	(bitb)	517 (74)	348 (17)	0.67
9-1	(tpu)	66	34	0.52
10-1	(prolog)	128	66	0.51
11-1	(diff)	149	57	0.38
12	(test)	7823 (648)	3890 (218)	0.50

も UtiLisp 32 を超える性能を示しているから，実用的な速度は達成しているということはいえるであろう．

7.2 移 植 性

7.2.1 移植の問題点

UtiLisp/C は SPARC 上で動かすことを目的として作られたが，多倍長整数を扱うルーチンを除けば，ほとんどC言語で書いてある．そのため，他のプロセッサ上でも少々の問題を解決すれば，多倍長整数を除いた仕様で動かすことができる．

移植の際に問題になるのは次のような点である．

●タグ付き演算命令

今のところ，タグ付き演算命令を用意しているのは SPARC だけなので，それ以外のプロセッサに移植するには，代わりにタグをチェックしてから演算するマクロまたはインライン関数を使う．タグ付き演算命令を持つプロセッサが出現した時は，アセンブリ言語命令をC言語から使う方法がコンパイラによって異なるため，それに対応しなければならない．

●ワード境界トラップ

プロセッサにワード境界からずれたアクセスによるトラップ機能がない場合は型チェックにこれを使うことができない．その場合は明示的な型チェックを行う必要がある．ワード境界トラップ機能があっても，signal ハンドラの中から原因となった Lisp オブジェクトを特定することができなければ，型チェックには使えない．

●インクリメンタルロード

UtiLisp/C のコンパイラは，Lisp のプログラムの翻訳結果としてC言語のコードを出力して外部のCコンパイラでコンパイルして，オブジェクトファイルを生成した後，それをメモリ上にロードするようになっている．つまり動的リンクの機構が必要になっているのである．動的リンクは，BSD 系の unix では用意されているが，System V 系の unix には用意されていない場合もある．その場合は，ld の代替品を用意しなければならない．また，動的リンクが用意されていても，オブジェクトファイルのフォーマットが異なる場合もある．この場合も変更が必要になる．

このように，処理系のソースプログラムを変更せずに移植ができるというわけではないが，慣れれば移植作業は数時間で済む．

実際に移植を試みたのは，MC 68020，80386，メインフレーム，R 3000 などである．移植の一例として R 3000 への移植を次項で述べる．

7.2.2 R 3000 への移植

RISC News などに使われている R 3000 は MIPS 社が設計開発した RISC プロセッサで，多くの計算機に採用されている．

R 3000 プロセッサは SPARC 同様ワード境界をまたがるアクセスを行うとバスエラーを生ずる．これを利用して組込み関数での型検査を省略することができる．しかし，SPARC と違ってタグ付き演算命令は用意されていない．そのため固定長整数演算の際には型チェックを必要とする．

MIPS 社の提供する R 3000 用のCコンパイラはグローバルオプティマイズを行う．組込み関数を末端関数化すると最適化の役に立つので，SPARC 同様引数エラーを signal を使って検出するようにした．

移植の際には，Cコンパイラの出力するオブジェクトコードのフォーマットの違いから，コンパイルされたコードのロード部分で書き直しが必要になったが，

表 3 R3000 におけるベンチマーク表
Table 3 Benchmarks on R3000.

番号	プログラム名	MIPS 1/60 sec	時間比 (対 SS1)
1-1	tarai	239	0.57
1-4	tak	136	0.58
2-1	list-tarai	46	0.49
2-2	srev	143	0.48
2-4	qsort	152	0.48
2-5	nrev	113	0.37
2-6	reverse	27 (26)	0.56
2-7	nreverse	10 (25)	0.45
3-1	string-tarai	483 (60)	0.54
4-1	flo-tarai	263 (17)	0.56
5-1	bubblesort	80	0.53
6-1	sequence	12	0.63
7-1	(bita)	45	0.48
7-3	(bitb)	146 (8)	0.42
10-1	(prolog)	31	0.47
11-1	(diff)	34	0.60

他はほとんどヘッダファイルの変更に留まった.

20 MHz の R 3000 を使った RISC News 上での実行速度は，**表 3** のようにテストによってばらつきがあるもののおおむね Sparc Station 1 の 1.6 倍から 2.7 倍である．タグ付き演算命令の使えない点で不利だと予想される tarai の場合でも，SPARC の 0.57 倍の時間で実行が終了する.

これは，Lisp のように再帰を多用する言語の実現にはレジスタウィンドウを使っている SPARC は向いていないということを表しているかもしれない．プロセッサと Lisp の相性については今後の検討が必要だろう.

8. 結　論

高級言語で処理系を記述する場合，一般的には実行するプロセッサに依存した最適化は困難である．そのため，アセンブリ言語で書いた場合と比較して，プロセッサの性能を出し切ることができず，実行効率が落ちるのが普通である.

UtiLisp/C は，移植性を考慮してC言語で記述されているにもかかわらず，SPARC プロセッサ上で使う時は，SPARC プロセッサの特徴を生かすように作られている．その結果，十分な速度を得ることができた.

さらに，同じような特徴をそなえた RISC プロセッサ上へも容易に移植することができ，高い性能を示すことが確認された．本論文に述べた手法は UtiLisp だけでなく RISC プロセッサ上で類似の言語処理系を開発する際に広く利用できるものと考える.

謝辞 和田英一教授および寺田 実，岩崎英哉，湯浅 敬，小西弘一，白川健治，村松正和の各氏の協力に感謝する.

参 考 文 献

1) 近山 隆：Lisp 処理系の構成法に関する研究，1981 年度東京大学工学部情報工学博士論文 (1982).
2) 近山 隆：UtiLisp システムの開発，情報処理学会論文誌，Vol. 24, No. 5, pp. 599-604 (1983).
3) 和田英一，富岡 豊：UtiLisp の MC 68000 への移植，情報処理学会記号処理研究会資料，SYM 29-3 (1984).
4) Kaneko, K. and Yuasa, K.: A New Implementation Technique for the UtiLisp System, Preprints of WGSYM Meeting, IPS Japan, SYM 41-7 (1987).
5) Muchnick, S. S.: SPARC の最適化コンパイラ (日本語訳)，ユニックス・マガジン，Vol. 4, No. 1, pp. 87-102 (1989).
6) 和田英一：UtiLisp/C の bignum ルーチン，情報処理学会記号処理研究会資料，SYM 53-4 (1989).
7) 田中哲朗：UtiLisp/C のインタプリタ，情報処理学会記号処理研究会資料，SYM 53-3 (1989).
8) 村松正和：UtiLisp/C のコンパイラの製作，情報処理学会記号処理研究会資料，SYM 53-5 (1989).

(平成 2 年 12 月 18 日受付)
(平成 3 年 2 月 12 日採録)

田中 哲朗 (正会員)
1965 年生．1987 年東京大学計数工学科卒業．1989 年同大学大学院工学系研究科情報工学専攻修士課程修了．現在同博士課程在学中．記号処理言語，漢字フォントに興味を持つ．ACM 会員.